4. USING M-API

This section explains the use of M-API to access arrays, tables, metadata. The user should refer to Section 6 for a discussion of the example programs included in the M-API distribution. In the following text the M-API routines names are from the C reference followed by the FORTRAN in parentheses, [i.e., **openMODISfile (OPMFIL)**].

4.1 Preparation

In order for M-API to be available for development programs the user will need to have his operating environment or makefiles reflect the locations of the M-API include files and the appropriate library. This being done, the following algorithm will serve as a starting point. In the following sections the **put** and **get** in bold refer to using the put or get function for the appropriate data object (arrays, groups, tables).

Step 1	Open the file	using **openMODISfile (OPMFIL)**
Step 1a	(Prepare data)	see below
Step 2	get/put data	see below
Step 3	Close the file	using **closeMODISfile (CLMFIL)** (Use a call to **completeMODISfile (CPMFIL)** instead when a NEW file has been created.)
Step 4	on error (or not) check the LogStatus file designated in the job's Process Control Files (PCF).	

This is an oversimplification but it provides a high-level view of M-API use in an application program. The next sections will deal with Steps 2a and 2, preparing/getting/putting data to/from the MODIS file.

The first step in writing an array to a file is to create a new file or open an existing file for read/write access (using **openMODISfile [OPMFIL]**). The *'modfil'* variable (C structure or FORTRAN array) returned by this routine is used to identify the file for all subsequent access operations.

In the last step the file must be closed after all access is completed and prior to program termination using **completeMODISfile (CPMFIL)** [for a new MODIS file] or **closeMODISfile (CLMFIL)** [for a pre-existing file]. Some data may not be written to the file until this routine is performed.

4.2 Data Groups

Data groups are synonymous with HDF Vgroups. A Vgroup in HDF is defined as "a structure designed to associate related objects" (HDF User's Guide, NCSA, 4/17/96).

4.2.1 Introduction to Data Groups (Vgroups)

Vgroups can be conceptualized as being similar to directories (or folders) on a computer file system. They provide a convenient way of associating disparate data

types, or data objects, even other Vgroups. In general Vgroups have two attributes that can be used for searching and classifying them: *name* and *class.* The Vgroup name is a character string that describes and references a particular data group. The Vgroup class is also a character string that further classifies and describes the data group. In M-API the class string can be written into the file with put, but is ignored otherwise.

4.2.2 Using Data Groups

Step 1a call **createMODISgroup (CRMGRP)**

M-API provides a function for creating a new data group **createMODISgroup (CRMGRP)**. This function should be called prior to the creation of related data objects. Then the appropriate function call to **put** the array or table will place the data in the file in that particular data group (see the following section on how to **put** data in to a MODIS file). For existing files, arrays and tables can be retrieved from a particular group with the appropriate call to **get** the particular data object. For cases where no group exists, simply set the *grpnam* variable to '\0'. Example 2 in Section 6.2 shows how to call **createMODISgroup**. Example 2a in Section 6.3 shows how to get data from a particular group.

4.3 Accessing Arrays

Nearly all MODIS products at L1 and L2 will make extensive use of multi-dimensional arrays for storing data. HDF supports arrays with a wide range of data types and sizes, and M-API offers considerable flexibility in defining, writing, and reading arrays. The following sections provide a general introduction to the use of arrays with the M-API and give specific usage and examples for accessing arrays.

4.3.1 Introduction to Arrays

Arrays are multi-dimensional, numerical data objects which are the same data type stored within HDF files. In HDF nomenclature, an array is referred to as SDS. A given file can store a large number of individual arrays, although as a general guideline the total number of data objects in a file should not exceed 100. Each array can be defined, written, and read independently of other arrays and data objects in a file. An array can store variables of any M-API supported data type except character string.

The essential elements of an array are as follows:

- Name - This is a character string which is used in all of the M-API array routines to designate the particular object. The name is stored in the file and can also be used by standard HDF tools to access the array.
- Data Type - Supported data types are signed and unsigned 8-bit, 16-bit, and 32-bit integers; 32-bit and 64-bit floating point.

- Rank - The number of dimensions for the array. A linear array has a rank of 1, a rectangular ("flat") array has a rank of 2, a cube has a rank of 3, and so on. HDF supports arrays with ranks as high as 32 dimensions, but use of higher-dimensional arrays should be considered carefully since they can be confusing to data users; if one of the dimensions is small it may be preferable to generate multiple arrays with fewer dimensions.
- Dimensions - The size of the array in each dimension. The ordering of dimensions in the M-API C and FORTRAN interfaces follows the conventions for these languages: the first dimension varies slowly in C and rapidly in FORTRAN.

In addition, the M-API supports several optional elements which can be used to describe or document an array.

- Array attributes - These are data which can be stored in the file as documentation for the array. Each attribute has a text name associated with one or more numerical values or a text string. Two frequently used attributes for text string data are 'long_name' and 'units'.
- Dimension attributes - These are identical to the array attributes, in function, but are associated with an array dimension instead of an entire array. Attributes and the M-API routines for accessing them are discussed in Section 4.5, "Accessing Metadata".
- Data Groups - HDF allows data objects to be organized hierarchically within a file by defining them as members of predefined data groups ('Vgroups'). The M-API supports the definition of data groups, and arrays can be associated with data groups at the time the array is created. Currently no conventions have been established for the use of data groups for MODIS data products.

The M-API provides flexible access to arrays. Once an array has been defined, the entire array or any desired subset can be read or written during each access. The text name of an array is used to identify it. Data may be written to the array and then subsequently read back into memory, if desired. HDF automatically fills any array elements which are not written to. (The default "fill value" is stored in a standard array attribute "_FillValue" that may be retrieved or changed using M-API routines).

In addition, arrays from multiple files can be accessed concurrently (e.g., reading an array from one file while writing to another). The files are distinguished by use of the M-API file handle variables (C structure or FORTRAN array) which the file opening routine **openMODISfile (OPMFIL**) returns.

Table 4-1 shows the array Interface modules. The specific M-API routines used for array access are discussed in Section 4.3. The syntax of these routines is described in detail in Appendix B-10. Some examples of their use are given in Examples 1, 2, 2a, 4, and 5.

Table 4-1. M-API Array Interface Modules

C	FORTRAN	Description
createMODISarray	CRMAR	Initializes an array structure in a file.
getMODISardims	GMARDM	Retrieves dimensions of an array structure.
putMODISarray	PMAR	Writes a subarray into an array structure.
getMODISarray	GMAR	Reads a subarray from an array structure.
putMODISarinfo	PMARIN	Writes an array attribute.
getMODISarinfo	GMARIN	Reads an array attribute.
putMODISdiminfo	PMDMIN	Writes an array dimension attribute.
getMODISdiminfo	GMDMIN	Reads an array dimension attribute.
putMODISdimname	PMDNAM	Writes a dimension name.
getMODISdimname	GMDNAM	Reads a dimension name.

4.3.2 Reading Arrays

Step 1a call **getMODISardims (GMARDM)** (optional)

Step 2 call **getMODISarray (GMAR)**

Use **getMODISardims** to retrieve the rank, dimensions, and data type of an array prior to getting it.

Proper dimensioning of the variable *dimsizes* (see Appendix D) to provide sufficient elements for the dimensions of the array structure may at first appear to require precognition. The easiest solution is to provide a generous (32 element) *dimsizes* array. Another approach is to use the *rank* variable as an input containing the number of elements in *dimsizes.* If *dimsizes* is inadequate for the multi-dimensional array structure in question, **getMODISardims (GMARDM)** will fail gracefully but will return the rank of the array structure, allowing for the dimension information to be retrieved with a second call.

4.3.3 Writing Arrays

Step 1a call **createMODISarray (CRMAR)**

Step 2 call **putMODISarray (PMAR)**

· Defines the array name with **createMODISarray.** The array must be created before it is possible to write data or attributes to it. Arrays may be given names up

to 256 characters long and ranks up to 32 dimensions. *Once an array is created, however, its name, data type, rank, and dimension length cannot be changed.*

- Writes data to (all or any part of) the array with **putMODISarray**; given starting indices inside the array and the dimensions ('edges') of the data to be written. **putMODISarray** may be called multiple times to write (or overwrite) data into the array.

The user must ensure that the rank, dimensions, and data type used to write the array are consistent with the array definition. To store a data "slab", the initial write location in the array and the length of each edge along each array dimension must be identified to define the region of the array to be written to. This is illustrated in Figure 4-1.

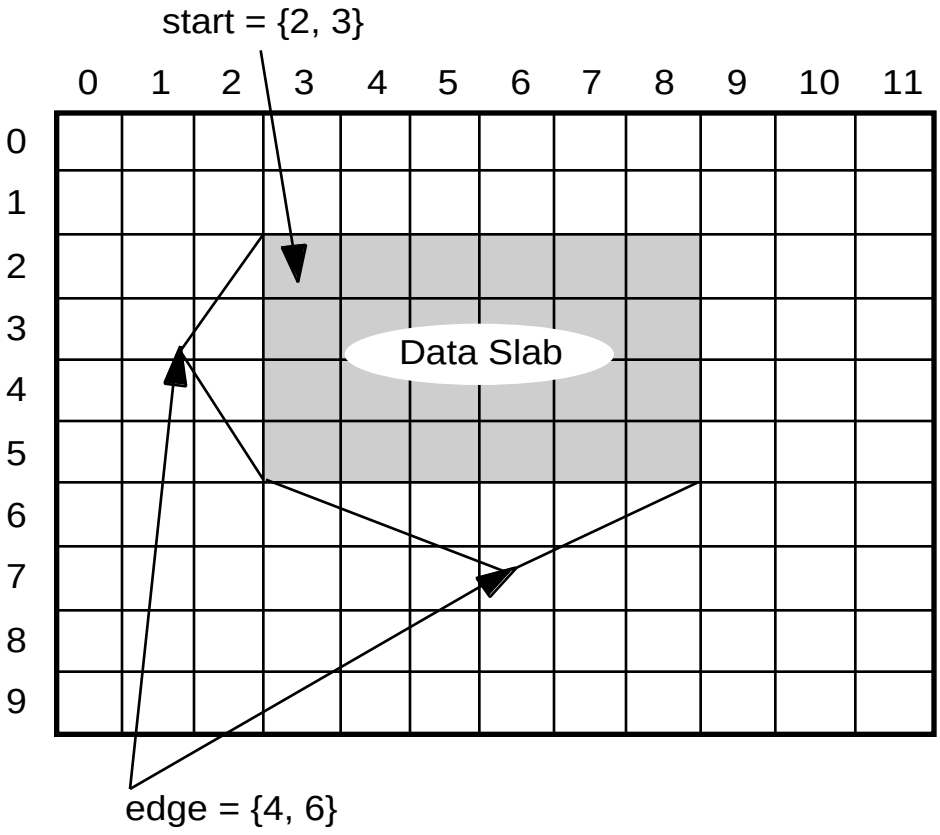

Figure 4-1. Writing Data into a Two-Dimensional Array

The argument *start* is an array specifying the location in the array where the first of the data values will be written. The *start* array must have a value for each dimension in the SDS. Each value must be smaller than its corresponding array dimension.

The argument *edge* is an array specifying the length along each array dimension that data values will be written to. The *edge* array must have a positive value for each

dimension in the SDS. The sum of each value in the *edge* array and the *start* array must not exceed the corresponding array dimension.

The region to write the data is defined by the values in the *start* and *edge* arguments to **putMODISarray**. In the example illustrated above, the first elements in the *data* buffer will be written into the array.

4.3.4 Array Attributes

Attributes and the M-API routines for accessing them are discussed in Section 4.5, "Accessing Metadata". However, for completeness consider when one wishes to read array attributes, or when one wants to create/modify attributes of existing arrays. M-API provides the following routines to aid the developer.

Two routines read attributes associated with an array:

Step 1a/2 call **getMODISarinfo (GMARIN)**

or call **getMODISdiminfo (GMDMIN)**

Two routines create or modify attributes associated with an array:

Step 1a/2 call **putMODISarinfo (PMARIN)**

or call **putMODISdiminfo (PMDMIN)**

4.4 ACCESSING TABLES (VDATA)

This section presents a discussion of M-API table structures. The concept of the table structure is explained, a general description of how a table structure is implemented in an HDF file is provided and the M-API routines for accessing them are introduced. Finally, there is a discussion of some problems that may be encountered when using the M-API table routines to access HDF Vdata that were not made using M-API.

4.4.1 Introduction to Tables

Conceptually, a table is a set of one or more records, with each record having an identical structure, and with annotation included to describe that record structure. Each column of the table (or HDF 'field') is assigned one of the permitted M-API number types (i.e., int16, float32,...) (see Table B-2 in Appendix B), and a text string field name. These field names and the data table name should be M-API-supplied constants. A table's record structure may contain several varieties of number types, in any order, and may be up to 32,767 bytes wide, but each record in the table has an identical size and structure to the others.

M-API utilities use the HDF Vdata structure to store tables. A Vdata provides the facility to store a set of records, each identical in structure. Each field in the record has a

unique text string name to identify it and may be of any standard number type, but has a fixed length. Vdata records are stored in a 'packed' format (i.e., there are no padding or alignment bytes stored in the record). A Vdata record is stored in precisely the amount of memory required to hold the sum of its constituent fields. As will be discussed later, this can cause problems when a program's data structure (which may have padding or alignment bytes) is used to write data to or read data from a Vdata.

Each Vdata must have a text name and a text class. The M-API identifies Vdata by name and requires all Vdata it creates to be named. The class name may be used to group a set of tables into a 'class'. M-API provides no direct means to search for a Vdata by class name.

All Vdata records created with M-API are stored in contiguous memory in the HDF file. This is the default HDF Vdata storage mode and it permits the M-API to append additional records to an existing Vdata. Therefore, all the data for a particular record must be in contiguous memory in the I/O buffer (and packed, without any padding bytes). When reading table records, the data retrieved from a particular record will be found packed in contiguous memory in the I/O buffer.

The M-API utilities provide the facility to create and read data table structures into MODIS HDF files (see Table 4-2). Creation of a table structure establishes it with an immutable table name, class name, Vgroup location, field name for each column, and record structure. Data are written to a table structure by records and may be overwritten. Reading from a data table structure is more flexible. A specific set of fields from a specific contiguous set of records may be read with a single call. Detailed information about the individual M-API routines that perform these functions is provided in Appendix B.

Table 4-2. M-API Table Interface

C	FORTRAN	Description
`createMODIStable`	`CRMTBL`	Initializes a table structure in a file.
`putMODIStable`	`PMTBL`	Writes data records into a table structure.
`getMODISfields`	`GMFLDS`	Retrieves dimensions of a table structure.
`getMODIStable`	`GMTBL`	Reads data records from a table structure.

4.4.2 Reading Tables

Step 1a call **getMODISfields (GMFLDS)** (optional)

Step 2 call **getMODIStable (GMTBL)**

Prior to reading from a table structure it is recommended (but not required) that **getMODISfields (GMFLDS)** be called to retrieve dimensional information about the table structure. This routine can provide the number of records in the table structure, the number of fields in each record, and strings listing the field headers, their data types, and the table structure's 'class', if any. In C, the data-type and number of records information may be used to dynamically allocate a retrieval buffer, as will be shown in an example later in this section. **getMODISfields** output arguments for which no information is desired may be set to NULL (this option is not available in FORTRAN).

The M-API utility **getMODIStable (GMTBL)** provides more flexibility for data retrieval than is permitted for data storage. Data may be read from any set of contiguous records with a single call. In addition, it is not necessary to read out entire records or to read out table structure fields in the order that they are stored in. The order that the desired fields are listed in the field name string input to **getMODIStable** will be the order that the data are listed in the buffer for each record. This will be demonstrated in an example later in this section. The ability to read a subset of record fields permits the addition or rearranging of fields in a new version of the software that writes the table structure without requiring programs that read earlier versions of the table to change their I/O, providing the fields of interest still exist in the new table. Matching the field name strings exactly to those stored in the table structure is required to prevent the data retrieval from aborting. Using M-API-supplied constants helps prevent such failures from occurring.

4.4.3 Creating and Writing Tables

Step 1a call **createMODIStable (CRMTBL)**

Step 2 call **putMODIStable (PMTBL)**

A MODIS HDF table structure is created using the **createMODIStable (CRMTBL)** utility. The file data structure (opened for writing or appending), a table name, the number of fields in the table, a string of field headers, and a string of data type strings are required inputs. The table name is the ASCII string M-API utilities will use to identify the table structure. It must be unique in the file and should be a M-API-supplied constant. The string of field headers is a comma-delimited set of names for each field in the order that the fields will appear in each record. A three-field headers string might look like "Header1,Header2,Header3", for example. Attention to the absence of spaces and of character case in field header strings is essential, since these header names are used to identify table fields. The data type string is also a comma-delimited string providing the number type for each field in order. This might appear for two floats and an integer field as "float32,float32,int16" (or 'REAL*4,REAL*4,INTEGER*2' in FORTRAN).

Each field in a M-API table structure record contains a single numerical value or character. However, HDF Vdata may be created with record fields that are fixed-length vectors. The M-API does not support the creation of such multi-element fields, nor is it

possible to identify a field as multi-element with M-API. M-API is capable of reading and writing records that contain multi-element fields. Each record in the table structure must still have identical structure and length, and each field must be of fixed length in every record.

The class name field and data group name arguments of **createMODIStable (CRMTBL)** are optional inputs. The class name provides the option to identify the table with a larger 'class' of data objects. Providing a data group name places the new table structure into a data group (HDF Vgroup); a 'subdirectory' of the MODIS HDF file containing associated data objects. Setting these parameters to NULL (or a string of blanks in FORTRAN) rejects these options.

Writing data to a table structure using **putMODIStable (PMTBL)** requires assembling the data to be stored (possibly of different types) into a buffer. The data must be ordered in the output buffer so that all data for each record is contiguous, and is in the order of the fields assigned in **createMODIStable (CRMTBL)** like so: {record0-field1,record0-field2,....record1-field1,record1-field2, etc.}. The buffer must contain an integral number of records to store with data for every field in each record.

The example programs (6, 7, and 8) shown in Section 6 demonstrate using the M-API table utilities. The first two programs create a new MODIS HDF file, create a table structure to hold the table shown below, and store *in situ* fabricated data in it. The next two programs then demonstrate how to read a record from this table structure. Possible approaches are shown for handling buffers used to transfer data to and from the table structure.

(A data group to place the table structure in would first have to be created using **createMODISgroup (CRMGRP)** before creating the table structure.) The header string and the data type string define the name, size, and placement of each field in a record. See Example 2 for a simple case using C.

The **putMODIStable (PMTBL)** routine identifies the target table structure merely by its name and the data group string set to NULL. Note that a CHARACTER string set to blank serves the same purpose in FORTRAN as the NULL does in C.

Calls to 'memcopy' are used in the C program to perform the same task as the EQUIVALENCE statements in the FORTRAN77 example. Another approach, using pointers into the data buffer is demonstrated in the example code for putMODIStable in the Section 6.

WARNING: As natural as it might appear, a structure of two floats and an integer should not be used to write to this table. C data structures may contain padding or alignment bytes which would make the structure longer than the 12 byte length of each Vdata record. Worse, such padding is machine and compiler dependent, so even if this problem could be corrected for structures on one platform, the solution probably would not port to others. If all the fields are the same data type, however, a contiguous array of that data type could be used for the output buffer.

4.4.4 Potential Problems with One Record Vdata's

The M-API table routines are designed to work together to provide a consistent interface to HDF Vdata. Problems may arise, however, in so-called 'mixed-mode' operations where a Vdata is accessed using both HDF and M-API routines. In particular, problems ranging from warning messages to outright loss of data may occur when a Vdata with a single data record is written without **putMODIStable (PMTBL)** and then subsequently accessed with M-API routines. These problems may be resolved in a future M-API version.

The following problems will only occur when a Vdata has a single record that was not written using M-API. **getMODIStable (GMTBL)** will successfully read the record (or any subset thereof), but will write a warning message that a 'dummy data record' was read. This warning is generated because the M-API has mistakenly identified the data record with a 'dummy' record M-API produces when it first creates a table structure. The record is not properly marked for the M-API to recognize it as real data. **getMODISfields (GMFLDS)** suffers a similar problem in that it will report that there are '0' data records in the table structure. The erroneous action taken by **putMODIStable (PMTBL)** in such circumstances is more serious. Mis-identifying the record as a dummy record, it will incorrectly overwrite the record if instructed to append data to the table structure.

There are several ways to avoid these problems. The most obvious is to use M-API routines for all Vdata creations and writes or at least perform all data writes to the Vdata with **putMODIStable (PMTBL)**. This may require a dummy record to be written to a newly created Vdata, just as **createMODIStable (CRMTBL)** does. The problem also may be avoided by writing the two data records to the Vdata without using M-API. M-API assumes that a table structure with more than one record no longer has a dummy record and so will not attempt to overwrite it.

4.5 ACCESSING METADATA

4.5.1 Types of Metadata

ECS has defined metadata as "all descriptive information which will accompany ECS standard data products" (ECS Core Metadata Standard, Release 2.0). The metadata covers all aspects of production processing through the final archive of the data with the GSFC DAAC.

ECS has identified the following set of metadata types:

- Processing-specific (e.g., the name of an input file),
- Quality Assurance (QA) related (e.g., statistical measures of product accuracy),
- Product-level (e.g., spatial bounds covering the product), and

· Inventory (e.g., ECS granule identifiers).

4.5.2 Metadata Processing in the ECS

The source of the metadata types described in Appendix B will be the production processing, with inputs from both the science Product Generation Executive (PGE) and the ECS process control system. The SDP Toolkit library routines will facilitate the reception of metadata from these sources and their transfer to HDF files. At the heart of this implementation is the Metadata Configuration File (MCF), which stores values of the types defined above using a syntax based on the PVL. Table 4-3 defines the M-API metadata interfaces. The MCF tools will extract the metadata from the MCF, and an additional routine will utilize ODL calls to parse the streams into objects. The library will also be capable of writing the metadata objects to the HDF file for final archival in the DAAC.

Table 4-3. M-API Metadata Interface

C	FORTRAN	Description
`getMODISECSinfo`	`GMECIN`	Get ECS metadata.
`getMODISfileinfo`	`GMFIN`	Get MODIS global file attribbute = value pairs.
`putMODISfileinfo`	`PMFIN`	Put MODIS global file attribbute = value pairs.
`substrMODISECSinfo`	`SMECIN`	Parse ECS metadata substrings.
`completeMODISfile`	`CPMFIL`	Write ECS metadata to file and close file.

4.5.3 Reading and Writing Metadata with M-API

Step 1a/2 call **getMODISfileinfo (GMFIN)**
or call **putMODISfileinfo (PMFIN)**
or call **getMODISECSinfo (GMECIN)**

The M-API routines support the metadata-handling concepts developed by ECS as part of theSDP. M-API provides the capability to write granule-level metadata to HDF files as a set of global attributes. The M-API routine **completeMODISfile (CPMFIL)** will write the ECS metadata for a newly created HDF file, at the time the file is closed. The M-API routine **putMODISfileinfo (PMFIN)** will write a global attribute and its value to the MODIS HDF file, while **getMODISfileinfo (GMFIN)** will read a global attribute and its value from the file.

The required global metadata parameters for MODIS HDF files are described in "ECS Metadata Syntax: Granules." Items that are written for a particular L2 product should be read from a L1B file with one or more calls to **getMODISECSinfo (GMECIN)** and written using the SDP Toolkit met routines. Items that vary with the granule and require scientist input will be generated by the code and will be written with SDP Toolkit calls to include processing-specific information during ECS production.

4.6 Miscellaneous

This section contains the miscellaneous routines and rounds out our discussion of the M-API library. Several of these programs are used generically throughout the underlying M-API library but are deemed to be useful for a general developer. See Table 4-4 for the Miscellaneous M-API Functions.

The routines **openMODISfile(OPMFIL)** and **closeMODISfile (CLMFIL)** are the definitive functions required for opening and closing files accessed by M-API. The routine **completeMODISfile (CPMFIL)** is designed to be used solely for finishing off (and closing) newly (M-API) created files.

Table 4-4. Miscellaneous M-API Functions

C	FORTRAN	Description
`MODISsizeof`	`MSIZE`	Returns the number of bytes associated with the character string names of each of the permitted data types
`openMODISfile`	`OPMFIL`	Open the MODIS file.
`closeMODISfile`	`CLMFIL`	Close the MODIS file.
`completeMODISfile`	`CPMFIL`	Complete (and close) the MODIS file.

NOTE: **completeMODISfile** (**CPMFIL**) has a diferent calling sequence in M-API 2.1 from previous versions.

Make frequent use of the general M-API utility **MODISsizeof (MSIZE).** It behaves similar to the C 'sizeof' macro. It returns the number of bytes associated with the character string names of each of the permitted data types. If a comma-delimited string of number types is given to it as an argument (such as might be provided by calling **getMODISfields [GMFLDS]**) it will return the total number of bytes the string represents, so that **MODISsizeof (MSIZE)** ("int32,float32,int16") would return 10, for example. This is a useful routine to help dynamically allocate memory for reading in data, if necessary.

5. M-API-SUPPLIED CONSTANTS, NAMING CONVENTIONS, AND DESCRIPTIONS

As was mentioned in Section 4, all data objects are referenced by string names that uniquely identify that object. A set of M-API-global constants is supplied in the mapi.h include file as macros for C routines and in the mapi.inc include file as PARAMETER set values for FORTRAN routines. The mapi.h file also contains prototypes for M-API utilities and structure declarations. It is essential that any module calling M-API utilities include mapi.h or mapi.inc. Providing and managing these name constants is an essential ingredient of M-API. In addition to the mapi include files, product specific include files have also been written. These include files define the different product specific M-API constants, described in this section. The names of these include files are listied in Table 5-1. Appendix B lists either the metadata description or the metadata name and the corresponding M-API constant.

Table 5-1. M-API -Supplied Product Specific Include Files

C	FORTRAN	Group Name	Table
`mapi.h`	`mapi.inc`	global values	B-1 through B-3
`mapiL1A.h`	`mapiL1A.inc`	Level 1A	B-4
`mapiL1Bgeo.h`	`mapiL1Bgeo.inc`	Level 1B /Geolocation	B-5
`mapiatmos.h`	`mapiatmos.inc`	Atmosphere	B-6
`mapiland.h`	`mapiland.inc`	Land	B-7
`mapiocean.h`		Oceans	B-8

5.1 Data Type Constants

Data type constants are provided to label the data type of data contained in a data object. They are acceptable inputs for the data_type parameter of **createMODISarray (CRMAR)** and **createMODIStable (CRMTBL)**. They may be used as inputs to the **MODISsizeof (MSIZE)** routine, which returns the number of bytes of memory required to store the data type.

The M-API constants for describing data types are listed in Table 5-2. These are the only data types recognized by M-API. The constant column identifies the 'macro' or 'symbolic constant' provided by the respective C and FORTRAN M-API include files. The constant names are identical for both languages, but the actual string represented is different for each language to be more closely associated with the specific language's nomenclature. These are the only M-API-supplied constants that have different content for different languages.

Table 5-2. M-API Data Type Constants

Constant	C	FORTRAN
I8	"int8"	'INTEGER*1'
I16	"int16"	'INTEGER*2'
I32	"long int"	'INTEGER*4'
I64	"int64"	'INTEGER*8'
R32	"float32"	'REAL*4'
R64	"float64"	'REAL*8'
TXT	"char *"	'CHARACTER*(*)'
UI8	"unint8"	'UINTEGER*1'
UI16	"unint16"	'UINTEGER*2'
UI32	"unint32"	'UINTEGER*4'
UI64	"unint64"	

5.2 Metadata Constants

Metadata are descriptive information about a MODIS data product which are included with the data product. In MODIS HDF files these are stored in a *keyword = value* format where the *keyword* is an identifying string used to label and locate the *value* information. Additional information about the metadata that are found in MODIS HDF files may be found in Section 6: Accessing Metadata.

The M-API provides constants for all specified MODIS metadata labels. These are listed in Appendix B, as product specific include files. In addition, some metadata may take on only a small domain of valid values. M-API also supplies constants for these small sets of metadata values.

5.3 Data Object Constants

Every data object in a MODIS HDF file, array structure, table structure, and data group, must have an ASCII string name. In addition, these data objects will have additional annotations associated with them. Some array structures will have dimensions with labels. A table structure must have a 'field' name for each column and may have a class name. A data group may also have a class name. All of these name strings must be available on a global basis so that all the routines may use them in searches.

All specified MODIS data objects and their associated annotations are available as constants in the M-API include file. These are all listed in Appendix B, as product specific include files. (Note: Refer to Section 1.7 for more specific information). Annotations specific to a data object are listed (both in the appendix and in the include files) together with the data object constant, but many annotations (e.g., #define HTINM "Height [meters]") are more generic.

5.4 MODIS File Definitions

For a file to be a useful vehicle for transferring information between processes, its expected content and structure must be known by each party. Even in a MODIS HDF file, with its 'self-describing' data objects, a link has to be made between the abstraction of the name of an object, what the general structure of that object is, and the information content provided by the object. Documenting and disseminating a file's content and structure provides an agreed-upon file definition that is required for proper software development and maintainance of the interface between processes accessing the file.

A set of MODIS file definitions are maintained by the MODIS SDST team under configuration control. These are MODIS Data Product File Definition forms, describing the content and structure of the file in sufficient detail to allow easy access to its contents using the M-API (and other) utilities. There is a file definition for every MODIS data product. The file definitions and the conventions they define are an integral part of the M-API. Examples of file definitions can be found in Appendix F.

5.5 Structure of MODIS Data Product File Definition Forms

The following pages show an example of a MODIS data product file and the associated MODIS Data Product File Definition form describing that file. Figure 5-1 is a conceptualization of the file's contents. This file has seven array structure SDSs and a table structure (Vdata) called Geolocation, all assigned to a single data group (Vdata) called Geophysical Data. The file also contains the standard MODIS data product metadata. The structure and character of these data objects and their attributes (such as label and table field names) are described in the MODIS Data Product File Definition form.

A MODIS Data Product File Definition form is composed from a set of four file definition elements: a header file definition form, an SDS definition form, a Vdata definition form, and a Vgroup definition form. Templates for the file definition elements are included in Appendix F, File Definition Templates. Every MODIS Data Product File Definition form begins with a header file definition. The other elements are included as needed to describe the specific data objects in the file.

The structure of a MODIS Data Product File definition form is generally hierarchical, or resembling an outline. The header element of the form describes the general content of the file. It is followed by the other elements describing the major data objects

contained in the file. Vgroup definition forms are followed by the element forms for the data objects inserted into the data group it describes. Each of the other element forms in turn describes the content of a data object, its structure and its associated attributes (such as dimension labels and table headers). Note that the order that data objects are described in on a MODIS Data Product File definition form has no implication for the actual placement of data objects in the file itself. Knowing where data are located on a MODIS HDF file is not necessary. The M-API, taking advantage of HDF, can locate an object in a MODIS HDF file if only its name and object type (e.g., array or table) are known.

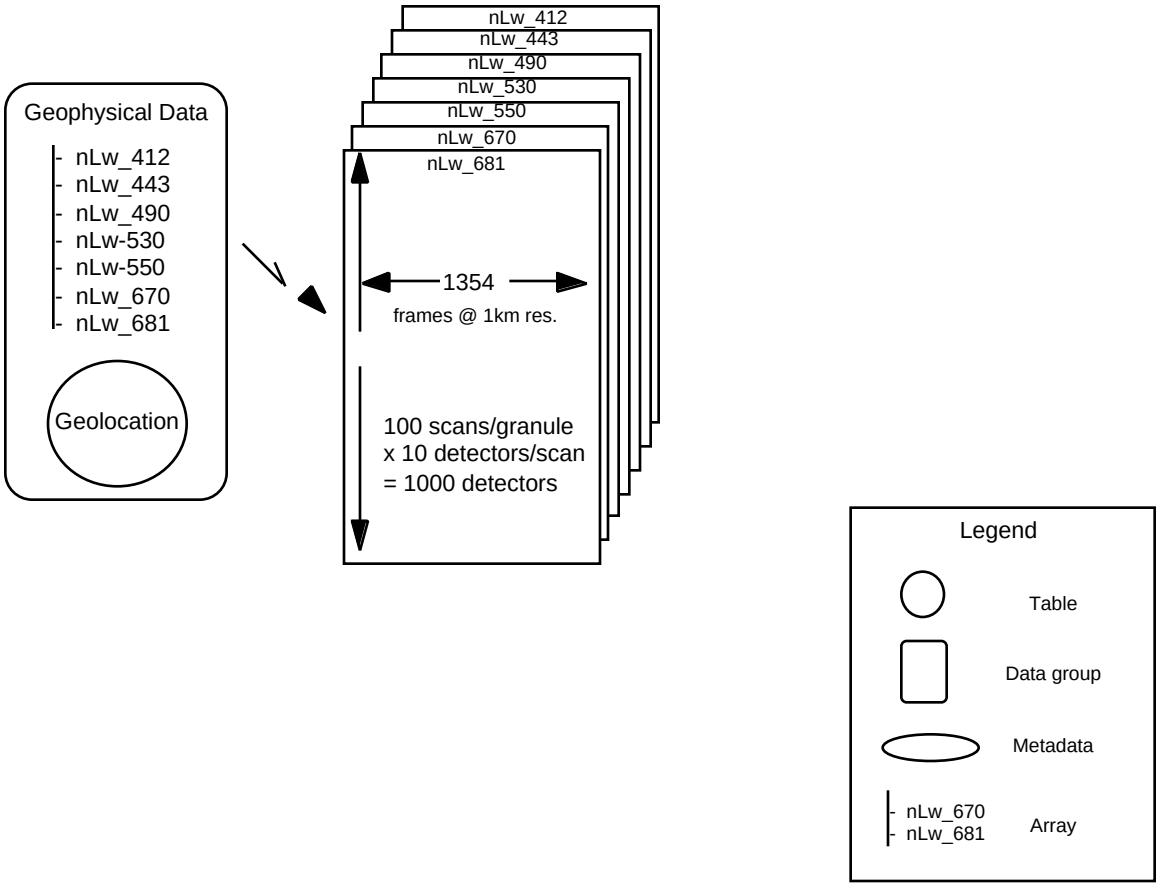

Figure 5-1. Sample Oceans (MOD18) HDF File Specification

All levels of a MODIS Data Product File definition share some of the same or similar information fields. The name of a data object is the most important. This character string is stored with the data object and is the key to M-API utility data object searches. Each form element has a description field. This is a sentence or two describing the

contents of a data object. The occurrence field provides for some flexibility in the structure of a MODIS HDF file. Some objects may not always occur in a particular MODIS Data Product File. Perhaps an array of data is pertinent only for daylight observations or a field in a table applicable only on a seasonal basis. The occurrence field will document under what circumstances a certain data object will or will not appear in the file. Each data element also has a contents field or fields. For the file itself and for Vgroups this is called 'Contents' and is simply a list of the objects that may occur therein.

A MODIS Data Product File definition also includes information about the subject file's metadata content. The header element includes a listing of all metadata labels that may occur in the file. The metadata 'Name' is the character string label that may be used in the M-API utility **getMODISfileinfo (GMFIN)** to retrieve metadata values or contents. If the metadata are not 'Always' present, the conditions of their presence or absence are provided in the Occurrence Column (examples of MODIS Data Product File definitions are shown in Appendix F)**.**

5.6 Maintenance and Location of the product specific M-API include files

When M-API is delivered, the current versions of the include files are included. In addition, these include files can also be found in the ftp site**: /projects/modis/util/modis_api/mapiheader** and in**: /cm/tools/src/MAPI2.0/h**.

In addition, a file (**mapi_include.status**) is maintained in these directories. Every attempt is made to keep the files at these locations as up-to-date as possible. However, the information contained in these files is gleaned from the HDF file specifications, thus when a file specification changes or one is added, the respective include file must be updated.

The macros contained in the file are never changed, only the metadata that it is associated with the macro is changed; though new macros may be added. Therefore it is crucial that the macro are used in place of the hard-coded metadata names. If a metadata name is changed, all places where it was hard-coded in the program will have to be changed, making sure that every occurrence is changed. On the other hand, if a macro is used then only the include file would have to be changed.

In the include file's prologue is a list of the file specifications that pertain to the include file. The file specification name, version, and revision date are listed. The beginning section contains common metadata that is used for all the products for that specific file. The rest of the file is divided into sections by product.

(This page intentionally left blank.)